

SDEV1201

Database Fundamentals

LESSON 18

Let's review

Anonymous Code Blocks

Why use DO?

Run multi-statement logic without creating a stored function/procedure.

Perfect for data fixes, quick loops, conditional DDL, and ad-hoc scripting.

Run multi-statement logic without creating a stored function/procedure.

Anonymous Code Blocks

--\$\$ create the block for the code that you will execute

DO \$\$

Declare --A section to declare variables

my_message TEXT:= 'Hello World!'; --Declares a variable called my_message and assigns 'Hello World to it'

my_mood TEXT:= 'groovy';

Begin--The executable statements go here

Raise Notice '% Hope you have a % day!', my_message,my_mood;

--Raises a message to pgamdins messages tab or whatever output console you are working in. It is mainly for testing and debugging.

--% is a place holder

--You can also format a raise Notice as

Raise Notice using message = my_message || ' Hope you have a ' || my_mood || 'day!';

End \$\$;

Variables

Variables

Variables are used to hold and manipulate values in memory. They must be declared before they are used, allowing PostgreSQL to allocate storage efficiently.

Syntax

Declare

```
variablename datatype;  
variablename datatype;  
variablename datatype;
```

All variables must be declared at the beginning of the block, before any executable statements.

Flow Control

You can control the execution of SQL statements using flow control structures such as If, Elself (optional), and Else (optional). You can have as many Elself statements as you want, but only one Else statement is allowed. You can use Begin and End for readability, but it is not required in If statements.

Flow Control

Syntax:

If **condition** Then

 Begin

 -- SQL statements

 End

Elsif **another_condition** Then

 Begin

 -- SQL statements

 End

Else

 Begin

 -- SQL statements

 End

End If;

Stored Procedures

A **stored procedure** is a **set of SQL statements**. It is very similar to other procedures that you may have used in other languages. The stored procedure has a name and can accept parameters.

A stored procedure in PostgreSQL is a named block of SQL statements that can be executed as a single unit. Stored procedures allow you to group related logic together, accept parameters, and control program flow; much like procedures in other programming languages.

They're compiled into machine language and stored in the database.

After it's been created, a SP can be executed without having to be re-compiled each time as a script would.

Stored Procedures

Simplified Syntax

In PostgreSQL, stored procedures are created using the CREATE PROCEDURE statement. They are written using the PL/pgSQL procedural language.

Syntax:

```
CREATE PROCEDURE procedure_name ( [parameter_list] )  
LANGUAGE plpgsql  
AS $$body$$  
BEGIN  
    -- SQL statements  
END;  
$$body$$;
```

Stored Procedures

Stored Procedure Capabilities

A PostgreSQL stored procedure can include:

- **DML statements** (INSERT, UPDATE, DELETE, SELECT)
- **Flow control** (IF, ELSIF, LOOP, etc.)
- **Transactions** (procedures can explicitly COMMIT or ROLLBACK)
- **Exception handling** (BEGIN ... EXCEPTION ... END)
- **Variable declarations and assignments**

Example

Invoking Stored Procedures

To execute a procedure:

```
CALL procedure_name ( [arguments] );
```

Example:

```
Call PR_UpdateWithdrawStatus();
```

Functions

If you need to return a value, you will want to use a function instead. Stored functions are created using the CREATE FUNCTION statement. They are also written using the PL/pgSQL procedural language.

Functions

Simplified Syntax:

```
CREATE FUNCTION function_name ( [parameter_list] )
```

```
Returns data_type
```

```
LANGUAGE plpgsql
```

```
AS $$
```

```
BEGIN
```

```
    -- SQL statements
```

```
    Return value;
```

```
END;
```

```
$$;
```

Functions

Invoking Stored Function

To run a function:

```
Select FN_GetTotalPayments();
```

Parameters

In PostgreSQL, parameters are declared inside parentheses after the procedure name.

Each parameter must have:

- A name (which starts with a colon or nothing at all — not @ as in SQL Server),
- A data type, and
- An optional default value can be assigned (for example, Default Null or = Null), which is used if no value is provided when calling the procedure or function.
- To test whether a parameter was passed, use Is Null, not = Null.

Example: Procedure with a Parameter

The following example creates a stored procedure that accepts a Student_ID parameter:

```
Create Or Replace Procedure PR_LookupStudent
(p_StudentID integer = Null)
Language plpgsql
As $body$
Begin
    If p_StudentID Is Null Then
        Raise Notice 'StudentID was not passed in';
    Else
        Raise Notice 'Looking up student with Id: %', p_StudentID;
    End If;
End;
$body$;
```


Executing a Stored Procedure with a Parameter

You can execute a stored procedure using the CALL statement:

```
Call PR_LookupStudent(12345);
```

Or, if you want to use the default parameter value: **Call**
PR_LookupStudent ();

Error Handling

Stored procedures in PostgreSQL can contain many different types of SQL statements including queries, Inserts, Updates, And Deletes. A DML (Data Manipulation Language) statement inside a stored procedure works exactly as it does outside of a stored procedure. However, there are some additional considerations when using DML statements inside stored procedures, especially when they are called from client applications (for example, Java, Python, or .NET). If an error occurs (such as a constraint violation or invalid data), PostgreSQL returns a detailed error message to the client. These messages are often technical and may not be meaningful to the end user. To make error handling more user-friendly, we can intercept any errors in our procedure and raise a clearer message.

Error Handling

Simplified Syntax:

Begin

-- Code that may cause an error

Exception

When condition Then

-- Handle the error

End;

Error Handling

Examples of handling possible exceptions:

Begin

-- Code that may cause an error

Exception

When too_many_rows Then

Raise Exception 'Duplicate record found error: %', SQLERRM;

When no_data_found Then

Raise Exception 'Record not found error: %', SQLERRM;

When unique_violation Then

Raise Exception 'Key value not unique error: %', SQLERRM;

When others Then

Raise Exception 'Unknown error: %', SQLERRM;

End;

Stored Function Exercises with Exceptions Solution

Question 1

Create a function named FN_GetCourseName that:

- Accepts a CourseID,
- Returns an error message if a CourseID is not passed in,
- Returns the course name if a record exists,
- Returns 'Course <input course id> Not Found' if no record exists.
- Test: `Select FN_GetCourseName ('PHYS2446');`

```
Create Function FN_GetCourseName (p_CourseID char(8) = null)
Returns varchar(40)
Language plpgsql
As $body$
Declare
    v_CourseName varchar(40);
Begin
    If p_CourseID is Null Then
        Return 'You must enter a CourseID.';
    Else
        Begin
            Select CourseName
            Into Strict v_CourseName
            From Course
            Where CourseID = p_CourseID;
            Return v_CourseName;
        Exception
            When No_Data_Found Then
                Return 'Course ' || p_CourseID || ' Not Found' ;
            When Others Then
                Return 'Unknown error retrieving course ' || p_CourseID || ' Error: '
            || SQLERRM;
        End;
    End If;
End;
$body$;
```

Question 1

Question 1 Testing

Select FN_GetCourseName ('PHYS2446');

Select FN_GetCourseName ('TEST1234');

Select FN_GetCourseName ();

Question 2

Create a function named FN_GetStudentAge that:

- Accepts a StudentID.
- Raises an error if a StudentID is not passed in,
- Raises an error, including the StudentID, if the StudentID passed in is invalid,
- Calculates and returns the student's age in years based on BirthDate.
- Test: `Select FN_GetStudentAge(199966250);`

Question 2

```
Create Function FN_GetStudentAge (p_StudentID integer = null)
Returns integer
Language plpgsql
As $body$
Declare
    v_BirthDate date;
    v_Age integer;
Begin
    If p_StudentID is Null Then
        Raise Exception 'You must enter a StudentID.';
    Else
        Begin
            Select BirthDate
            Into Strict v_BirthDate
            From Student
            Where StudentID = p_StudentID;
            v_Age = Date_part('Year', Current_Date) -
Date_part('Year',v_BirthDate);
            Return v_Age;
        Exception
            When No_Data_Found Then
                Raise Exception 'Student % not found.', p_StudentID;
            When Others Then
                Raise Exception 'Error calculating age for student %. Error: %',
p_StudentID, SQLERRM;
        End;
    End If;
End;
$body$;
```

Question 2 Testing

Select FN_GetStudentAge(199966250);

Select FN_GetStudentAge(777777777);

Select FN_GetStudentAge();

Question 3

Create a function named `FN_GetCourseEnrollmentCount` that:

- Accepts a `CourseID` and a `Semester`,
- Raises an error message if a `CourseID` or `Semester` are not passed in,
- Raises an error, including the `CourseID`, if the `CourseID` passed in does not exist,
- Raises an error, including the `CourseID` and `Semester`, if the `CourseID` passed in has no registrations for that semester,
- Returns the number of students registered in that course for that semester.
- Test: `Select FN_GetCourseEnrollmentCount('CPSC1012', '2000S');`

```

Create Function FN_GetCourseEnrollmentCount (p_CourseID char(8) = Null, p_Semester
char(5) = Null)
Returns integer
Language plpgsql
As $body$
Declare
    v_Count integer;
Begin
    If p_CourseID is Null or p_Semester is Null Then
        Raise Exception 'You must enter a CourseID and a semester.';
    Else
        If Not Exists (Select 1 From Course
                        Where CourseID = p_CourseID) Then
            Raise Exception 'CourseID % is not valid', p_CourseID;
        Else
            If Not Exists (Select 1 From Registration
                           Where CourseID = p_CourseID and
                           Semester = p_Semester) Then
                Raise Exception 'There are no registrations for CourseID % in Semester %',
p_CourseID, p_Semester;
            Else
                Begin
                    Select Count(CourseID)
                    Into v_Count
                    From Registration
                    Where CourseID = p_CourseID and Semester = p_Semester;
                    Return v_Count;
                Exception
                    When Others Then
                        Raise Exception 'Error retrieving class count % in %. Error: %', p_CourseID,
p_Semester, SQLERRM;
                End;
            End If;
        End If;
    End If;
End;
$body$;

```

Question 3

Question 3 Testing

Select FN_GetCourseEnrollmentCount('CPSC1012', '2000S');

Select FN_GetCourseEnrollmentCount('TEST1234', '2000S');

Select FN_GetCourseEnrollmentCount('COMP1017', '2005J');

Select FN_GetCourseEnrollmentCount();

Question 4

Create a function named `FN_CheckCourseSpace` that compares numbers of registrations in that course/semester with `MaxStudents` of the course.

- Accepts a `CourseID` and a `Semester`,
- Raises an error message if a `CourseID` or `Semester` are not passed in,
- Assumes the `CourseID` and `Semester` are valid if they are both passed in,
- If the course is at or above capacity, returns the string "Course < CourseID > and Semester < Semester > is full",
- If the course is under capacity, returns the string "Course < CourseID > and Semester < Semester > is available".
- Test: `Select FN_CheckCourseSpace ('DMIT2590', '2005M');`
- Test: `Select FN_CheckCourseSpace ('DMIT2015', '2002S');`

Question 4

```
Create Function FN_CheckCourseSpace (p_CourseID char(8) = Null,  
p_Semester char(5) = Null)  
Returns varchar(40)  
Language plpgsql  
As $body$  
Declare  
    v_max integer;  
    v_count integer;  
Begin  
    If p_CourseID is Null or p_Semester is Null Then  
        Raise Exception 'You must enter a CourseID and a semester.';  
    Else  
        Select Maxstudents  
        Into v_max  
        From Course  
        Where CourseID = p_CourseID;  
  
        Select Count(CourseID)  
        Into v_count  
        From Registration  
        Where CourseID = p_CourseID  
        and Semester = p_Semester;  
  
        If v_count >= v_max Then  
            Return 'Course ' || p_CourseID || ' and Semester ' || p_Semester || ' is  
full';  
        Else  
            Return 'Course ' || p_CourseID || ' and Semester ' || p_Semester || ' is  
available';  
        End If;  
    End If;  
End;  
$body$;
```


Question 4 Testing

Select FN_CheckCourseSpace ('DMIT2590', '2005M');

Select FN_CheckCourseSpace ('DMIT2015', '2002S');

Select FN_CheckCourseSpace ('');

Today's Objective

By the end of this section, you will be able to:

- ❑ Write **stored procedures with transactions**

Documentation can be found in Brightspace in the “Week 10 & 11 – Stored Procedures” section under Materials. “Stored Procedures with Transactions (PostgreSQL)”.

Transactions

Transactions are SQL structures that ensure that a **complete Logical Unit of Work (LUW)** is complete or fails. The key is understanding that each of these LUW consists of more than one step.

Let's consider an example of registering for a course:

1. Add a record to Registration table
2. Update student's BalanceOwing

What if step 1 worked and then step 2 failed? We'd have inconsistent data in our db!

Transactions

Another example.

Let's consider transferring funds from one bank account to another:

1. Subtract funds from 1st bank account – Chequing
2. Add funds to the 2nd bank account - Saving

What if step 1 worked and then step 2 failed? We'd have inconsistent data in our db!

Transactions

Using PostgreSQL transactions means all the operations to complete a LUW work or they all fail. So, if one of the operations to complete a LUW fails then all the operations to complete the LUW fail and it is like the LUW was never attempted in the first place. True, it did not make the registration work or the transfer of funds work but it did put the data back to the states it was in before it was attempted. Better than having some of the data altered and some not and therefore your data in the tables is in an unknown state! Keep in mind that what caused one the operations to fail might have been out of our control such as a network problem, hardware problem, or something else. So, even with perfect code and logic something can still happen.

Transactions

So, by defining a group of SQL DML statements in a transaction enable the server to maintain the integrity of the data. Ensuring all the statements that make up that transaction all succeed or all fail.

In PostgreSQL stored procedures (not functions) you can use Commit and Rollback to save or reverse any changes. A new transaction is started automatically after a transaction is ended using these commands, so there is no separate Begin Transaction command.

Transaction Syntax

<u>Statement</u>	<u>Description</u>
------------------	--------------------

Commit	Saves all changes made during the transaction
--------	---

Rollback	Reverses all changes made during the transaction
----------	--

The default behavior for the PL/pgSQL body is atomicity and if the body cannot complete, it is rolled back before entering the exceptions block. When an error is caught by an Exception clause, the local variables remain as they were when the error occurred, but all changes to persistent database state within the block are rolled back.

Transaction Example

```
Create Procedure PR_AddStudentPayment
(p_PaymentID integer = Null, p_PaymentDate date = Null, p_Amount decimal(6,2) = Null, p_PaymentTypeID integer =
Null, p_StudentID integer = Null)
Language plpgsql
As $body$
Begin
  If p_PaymentID Is Null Or p_PaymentDate Is Null Or p_Amount Is Null Or p_PaymentTypeID Is Null Or p_StudentID Is Null
Then
  Raise Notice 'You must enter a Payment ID, Payment Date, Payment Amount, PaymentTypeID, and Student ID';
Else
  Begin
    Insert Into Payment (PaymentID, PaymentDate, Amount, PaymentTypeID, StudentID)
    Values (p_PaymentID, p_PaymentDate, p_Amount, p_PaymentTypeID, p_StudentID);
    Raise Notice 'Payment % inserted successfully for student %', p_PaymentID, p_StudentID;
    Exception When Others Then
      Rollback ;
      Raise Exception 'Insert Payment % failed with Error: %', p_PaymentID, SQLERRM;
  End;
  Begin
    Update Student
    Set BalanceOwing = BalanceOwing - p_Amount
    Where StudentID = p_StudentID;
    Raise Notice 'Payment % applied successfully for student %', p_PaymentID, p_StudentID;
    Exception When Others Then
      Rollback ;
      Raise Exception 'Update Student % failed with Error %', p_StudentID, SQLERRM;
  End;
  Commit;
End If;
End;
$body$
```

Create a procedure PR_AddStudentPayment that inserts a payment record in the Payment table and reduces the Balance Owing value in the Student table. Use a transaction to ensure both succeed or both fail. Input parameters are Payment ID, Payment Date, Payment Amount, PaymentTypeID, and Student ID.

To Test And Run

-- Run with no input arguments

Call PR_AddStudentPayment ();

Select PaymentID, PaymentDate, Amount, PaymentTypeID, StudentID from Payment where PaymentID = 34;

Select StudentID, BalanceOwing from Student where StudentID = 200122100;

Call PR_AddStudentPayment (34, Current_Date, 450.00, 1, 200122100);

Select PaymentID, PaymentDate, Amount, PaymentTypeID, StudentID from Payment where PaymentID = 34;

Select StudentID, BalanceOwing from Student where StudentID = 200122100;

Activity

Work on “Stored Procedure Exercises with Transactions (PostgreSQL)” which is located in Brightspace “Week 10 & 11 – Stored Procedures” under Activities.

Question 1

Create a procedure PR_AddStudentWithPayment that adds a student with an initial payment:

Accept student information (StudentID, first name, last name, gender, birth date, balance owing) and payment information (PaymentID, PaymentTypeID, amount paid).

Insert a record into Student, then into Payment using the current date as the payment date.

If either insert fails, display appropriate error message and rollback both inserts.

Print success status with the first name, last name, and amount paid.

```
Create Procedure PR_AddStudentWithPayment(p_StudentID integer, p_FirstName varchar(25), p_LastName  
varchar(35),
```

```
    p_Gender char(1), p_BirthDate date, p_BalanceOwing decimal(8,2),  
    p_PaymentID integer, p_PaymentTypeID smallint, p_Amount decimal(8,2))
```

```
Language plpgsql
```

```
As $body$
```

```
Begin
```

```
    Begin
```

```
        Insert Into Student (StudentID, FirstName, LastName, Gender, BirthDate, BalanceOwing)
```

```
        Values (p_StudentID, p_FirstName, p_LastName, p_Gender, p_BirthDate, p_BalanceOwing);
```

```
    Exception
```

```
        When unique_violation Then
```

```
            Rollback;
```

```
            Raise Exception 'Insert Into Student failed with duplicate StudentID %, Error: %', p_StudentID, SQLERRM;
```

```
        When Others Then
```

```
            Rollback;
```

```
            Raise Exception 'Insert Into Student failed: %', SQLERRM;
```

```
    End;
```

```
Begin
```

```
    Insert Into Payment (PaymentID, PaymentDate, Amount, PaymentTypeID, StudentID)
```

```
    Values (p_PaymentID, Current_Date, p_Amount, p_PaymentTypeID, p_StudentID);
```

```
Exception
```

```
    When unique_violation Then
```

```
        Rollback;
```

```
        Raise Exception 'Insert Into Payment failed with duplicate PaymentID %, Error: %', p_PaymentID,
```

```
SQLERRM;
```

```
    When Others Then
```

```
        Rollback;
```

```
        Raise Exception 'Insert Into Payment failed: %', SQLERRM;
```

```
End;
```

```
Commit;
```

```
    Raise Notice 'Student % % and initial payment % recorded successfully.', p_FirstName, p_LastName,
```

```
p_Amount;
```

```
End;
```

```
$body$;
```

Question 1

Question 1 Testing

-- Note: you need to Cast the PaymentTypeID to smallint [1::smallint] to tell the database the value is not an integer

Call PR_AddStudentWithPayment(200400001, 'New', 'StudentID', 'F', to_date('Apr 01 2000', 'Mon DD YYYY'), 500.00, 40, 1::smallint, 100.00);
-- new student

Call PR_AddStudentWithPayment(200312345, 'Existing', 'StudentID', 'M', to_date('Dec 31 2000', 'Mon DD YYYY'), 400.00, 44, 1::smallint, 100.00); -- Duplicate key (StudentID)

Call PR_AddStudentWithPayment(200319999, 'Existing', 'PaymentID', 'M', to_date('Oct 31 2000', 'Mon DD YYYY'), 300.00, 24, 1::smallint, 100.00); -- Duplicate key (PaymentID)

Question 2

Create a procedure PR_TransferBalance that transfers payment between students:

Accepts from_id integer, to_id integer, amount decimal(8,2).

Decreases one student's balance and increases the other student's balance.

If either student doesn't exist or the balance is insufficient, rollback.

Print a success message otherwise.

Question 2

```
Create Procedure PR_TransferBalance (p_from_id integer, p_to_id integer, p_amount decimal(8,2))
Language plpgsql
As $body$
Declare
    v_from_balance decimal(8,2);
Begin
    If Not Exists (Select 1 From Student
                   Where StudentID = p_from_id) Then
        Raise Exception 'From student % not found.', p_from_id;
    Else
        If Not Exists (Select 1 From Student
                       Where StudentID = p_to_id) Then
            Raise Exception 'To student % not found.', p_to_id;
        Else
            Select BalanceOwing
            Into v_from_balance
            From Student
            Where StudentID = p_from_id;
            If v_from_balance < p_amount Then
                Raise Exception 'Insufficient funds for student % to transfer.', p_from_id;
            Else
                Begin
                    Update student
                    Set BalanceOwing = BalanceOwing - p_amount
                    Where StudentID = p_from_id;
                Exception
                When Others Then
                    Rollback;
                    Raise Exception 'From student % update failed. Error: %', p_from_id, SQLERRM;
            End;
            Begin
                Update student
                Set BalanceOwing = BalanceOwing + p_amount
                Where StudentID = p_to_id;
                Raise Notice 'Transferred $% from Student % to Student %.', p_amount, p_from_id, p_to_id;
            Exception
            When Others Then
                Rollback;
                Raise Exception 'From student % update failed. Error: %', p_to_id, SQLERRM;
            End;
            Commit;
        End If;
    End If;
End If;
End;
$body$;
```

Question 2 Testing

Call PR_TransferBalance(200312345, 199912010, 100.00);

Call PR_TransferBalance(999999999, 199912010, 100.00); -- Invalid To
StudentID

Question 3

Create a procedure PR_PurgeOldPayments that deletes payments older than a given date:

Accept a date

If no old payments are found, print a message and do nothing.

If old payments are found, delete and commit.

If an error occurs, rollback and print an error message.

Question 3

```
Create Procedure PR_PurgeOldPayments (p_cutoff_date date = Null)
Language plpgsql
As $body$
Begin
    If p_cutoff_date is Null Then
        Raise Notice 'You must enter a cutoff date.';
    Else
        Begin
            If Not Exists (Select 1 From Payment
                           Where PaymentDate < p_cutoff_date) Then
                Raise Notice 'No old payments found, no changes made.';
            Else
                Begin
                    Delete From Payment
                    Where PaymentDate < p_cutoff_date;
                Exception
                    When Others Then
                        Rollback;
                        Raise Notice 'Error Deleting old payments: %', SQLERRM;
                End;
                Commit;
                Raise Notice 'Old payments deleted successfully.';
            End If;
        End;
    End If;
End;
$body$;
```

Question 3 Testing

```
select * from Payment  
where PaymentDate < '2001-01-01'
```

Call PR_PurgeOldPayments('2001-01-01'); -- 5 records

```
select * from Payment  
where PaymentDate < '2001-01-01'
```

```
select * from Payment  
where PaymentDate < '2000-01-01'
```

Call PR_PurgeOldPayments('2000-01-01'); -- 0 records

Call PR_PurgeOldPayments(); -- No parameters

Question 4

Create a procedure PR_RegisterForCourses that registers a student in multiple courses:

Accept a StudentID integer, and three course IDs.

Insert one registration per course for the current Semester (2025F), Mark (Null), WithdrawYN (N), and Staff (1).

If any insert fails (invalid course/student, duplicate record, etc.) rollback all inserts.

Print final status.

Question 4 Testing

Call PR_RegisterForCourses (200312345, 'COMP1017', 'DMIT2028', 'DMIT2003');

Call PR_RegisterForCourses ();

Homework

Continue working on “Stored Procedure Exercises with Transactions (PostgreSQL)” and “Stored Function Exercises with Exceptions (PostgreSQL)”. Both are found in Brightspace in the “Week 10 & 11 – Stored Procedures” section under Materials. “Stored Procedures with Transactions (PostgreSQL)”.